

# Machine Learning Homework 6 Report

## Kernel K-means and Spectral Clustering

312554056 紀品榕

### I. Kernel K-means

#### A. Code explanation

##### a. Libraries and modules imported

Libraries and modules imported into the program are shown below. The numpy module is imported for mathematical operations on arrays. The scipy.spatial.distance module is imported to compute the distance between two matrices. The PIL library is imported to read images and visualize the algorithm's results. The time module is used for time counting.

```
import numpy as np
from scipy.spatial.distance import cdist
from PIL import Image
import time
```

##### b. The overall workflow of the program

The data path of the input image, the clustering number, the parameters used in the kernel,  $\gamma_s$ , and  $\gamma_c$ , and the initialization of k-means used in both kernel k-means and spectral clustering are set in each for loop. Then, the image was loaded to get the color and spectral information and the image's shape. The kernel of the image is computed and used in the clustering steps. The initial clustering chooses the center points as the method the user asked for and performs the initial clustering. Then, the initial result is used in kernel k-means to cluster until the result converges. The clustering result of each iteration is saved during clustering and visualized after the clustering is finished.

```
if __name__ == '__main__':
    gs=[1e-3,1e-4]
    gc=[1e-3,1e-4]
    img_list=["image1.png","image2.png"]
    for i in range(2):
        for clusterNum in range(2,5):
            for gs_idx in range(2):
                for gc_idx in range(2):
                    for m in range(2):
                        start = time.time()
                        img_c,img_s,rows,cols,channels = load_img(img_list[i])

                        kernel = calculate_kernel(img_s,gs[gs_idx],img_c,gc[gc_idx])
                        cluster_result = get_initial_cluster(m,clusterNum,kernel,rows,cols,img_s)

                        perform_kernel_kmeans(rows,cols,kernel,clusterNum,cluster_result,img_list[i],m,gs[gs_idx],gc[gc_idx])
                        end = time.time()
                        print("The time of mode ",str(m)," execution is :","{:.2f}".format((end-start)), "\n")
```

c. Data loading

The `load_img` function uses the `filepath` parameter to open the image with the PIL library. The row number, the column number, and the channel number are got and used to calculate the color matrix and the spectral and returned.

```
def load_img(filepath):
    img = Image.open(filepath)
    img = np.array(img)
    rows, cols, channels = img.shape

    img_c = img.copy().reshape((-1, channels))
    img_s = np.zeros((rows*cols, 2), dtype=int)
    for r in range(rows):
        for c in range(cols):
            img_s[r*rows+c] = [r, c]

    return img_c, img_s, rows, cols, channels
```

d. Kernel calculating

The defined kernel  $k(x, x')$  is computed below. It is the product of two RBF kernels.

$$k(x, x') = e^{-\gamma_s \|S(x) - S(x')\|^2} * e^{-\gamma_c \|C(x) - C(x')\|^2}$$

where  $S(x)$  is the coordinate of the pixel,  $C(x)$  is the RGB value of each pixel,  $\gamma_s$  is the hyper-parameter for the RBF kernel considering the spatial information, and  $\gamma_c$  is the hyper-parameter for the RBF kernel considering the color information.

```
def calculate_kernel(img_s, gs, img_c, gc):
    spacial_dis = cdist(img_s, img_s, 'sqeuclidean')
    color_dis = cdist(img_c, img_c, 'sqeuclidean')

    return np.exp(-gs*spacial_dis)*np.exp(-gc*color_dis)
```

e. Getting the initial centers

The `get_centers` function calculates the initialization of k-means clustering, and the `mode` parameter controls the different methods used for initialization, where mode 0 represents the random initialization, and mode 1 represents the k-means++ method. After calculation, the function returns the index of each center in the spatial matrix.

The random method randomly selects and returns three different indices of data coordinates. On the other hand, the k-means++ method initially chooses a single index of data coordinates and then repeats the process until it identifies a sufficient number of center points. This involves calculating the distance between the current centers and the current pixel and selecting a new center index based on the probability proportional to the distance from the closest center. As a result,

the point with the maximum distance from the nearest center point is more likely to be chosen as a newly added center point.

```
def get_centers(rows,cols,clusters,img_s,mode):
    pixels = rows*cols
    if mode == 0:
        return np.random.choice(pixels,clusters)
    elif mode == 1:
        center_list = np.zeros(clusters,dtype=int)
        center_list[0] = np.random.choice(pixels)

        for k in range(1,clusters):
            dis = np.zeros(pixels)
            for p in range(pixels):
                min_dis_value = np.Inf
                for k_idx in range(k):
                    current_dis_value = np.linalg.norm(img_s[center_list[k_idx],:]-img_s[p,:])
                    if current_dis_value < min_dis_value:
                        min_dis_value = current_dis_value
                dis[p] = min_dis_value
            dis /= np.sum(dis)
            new_center_idx = np.random.choice(pixels,p=dis)
            center_list[k] = new_center_idx
        print("center list", center_list)

    return center_list
else:
    print("No such mode. Perform K-means++ instead")
    return get_centers(rows,cols,clusters,img_s,1)
```

f. Getting the initial clustering

After getting the initial center point coordinate indices, initial clustering is performed. For each pixel in the image, the distance between the current pixel and each center point is calculated, and then the pixel will be categorized into the same cluster with the minimal distance.

```
def get_initial_cluster(mode,clusters,kernel,rows,cols,img_s):
    centers = get_centers(rows,cols,clusters,mode)
    pixels = rows*cols
    cluster_result = np.zeros(pixels, dtype=int)
    for idx in range(pixels):
        distance = np.zeros(clusters)
        for k in range(clusters):
            distance[k] = kernel[idx,idx]+kernel[centers[k],centers[k]]-2*kernel[idx,centers[k]]
        cluster_result[idx] = np.argmin(distance)

    return cluster_result
```

g. Performing kernel k-means

The *perform\_kernel\_kmeans* function is used to perform kernel k-means clustering. The maximum iteration number and the threshold are set. The initial clustering result is visualized by the *get\_current\_image* function and saved for later use. While the maximum iteration number is not reached yet, the iteration count parameter will be renewed, and the new clustering result is again calculated using the last cluster result and the precomputed kernel by the *get\_new\_cluster* function. If the L2 norm of the new and old clustering results is smaller than the threshold, the computation is seen as coverage and stop; otherwise, set the latest result as the old one and visualize and save it. After the clustering, the whole

clustering process is visualized and saved.

```
def perform_kernel_kmeans(rows,cols,kernel,clusters,cluster_result,filePath,mode,gs,gc):
    max_iter = 50
    threshold = 1e-2
    img_list = []
    img_list.append(get_current_image(rows,cols,cluster_result))

    iter = 1
    while iter < max_iter:
        print(iter)
        iter += 1
        new_cluster_result = get_new_cluster(rows,cols,clusters,cluster_result,kernel)
        if(np.linalg.norm((new_cluster_result-cluster_result))<threshold):
            break
        cluster_result = new_cluster_result.copy()
        img_list.append(get_current_image(rows,cols,cluster_result))

    title = filePath[:-4]+"_k_means_clusterNum_"+str(clusters)+"_mode_"+str(mode)
    title += "_gs_"+str(gs)+"_gc_"+str(gc)

    img_list[0].save(title+".gif",save_all=True,append_images=img_list[1:],optimize=False,loop=0,duration=300)
    img_list[-1].save(title+"_result.png")
    return
```

The `get_new_cluster` function follows the equation below to calculate the distance between a particular point  $x_j$  and its corresponding cluster center in the feature space to cluster.

$$\begin{aligned} \|\phi(x_j) - \mu_k^\phi\| &= \left\| \phi(x_j) - \sum_{n=1}^N \alpha_{kn} \phi(x_n) \right\| \\ &= k(x_j, x_j) - \frac{2}{|C_k|} \sum_n \alpha_{kn} k(x_j, x_n) - \frac{1}{|C_k|^2} \sum_p \sum_q \alpha_{kp} \alpha_{kn} k(x_p, x_q) \end{aligned}$$

Where  $x_j$  is the current image pixel,  $k(x, x')$  is the kernel,  $|C_k|$  is the total number of data points in the  $k$ -th cluster,  $\alpha_{kn}$  represents whether the data point  $x_k$  is assigned to the  $k$ -th cluster or not,  $\alpha_{kn} = 1$  means it is assigned to the  $k$ -th cluster. Each term in the equation is calculated separately and then added together. The data point  $x_j$  is assigned to the cluster with the smallest distance with it.

```
def get_new_cluster(rows,cols,clusters,cluster_result,kernel):
    pixels = rows*cols
    new_cluster_result = np.zeros(pixels, dtype=int)
    each_cluster_count = np.zeros(clusters, dtype=int)
    last_term = np.zeros(clusters)
    second_term = np.zeros((pixels,clusters))

    for k in range(clusters):
        each_cluster_count[k] = (cluster_result==k).sum()
    each_cluster_count[each_cluster_count==0] = 1

    for k in range(clusters):
        tmp_kernel = kernel.copy()
        for p in range(pixels):
            if cluster_result[p]!=k:
                tmp_kernel[p,:]=0
                tmp_kernel[:,p]=0
        last_term[k] = np.sum(tmp_kernel)/each_cluster_count[k]**2

    for p in range(pixels):
        for k in range(clusters):
            second_term[p,k] += np.sum(kernel[p,:][np.where(cluster_result==k)])
            second_term[p,k] *= 2.0/each_cluster_count[k]

    for p in range(pixels):
        distance = np.zeros(clusters)
        for k in range(clusters):
            distance[k] += kernel[p,p]-second_term[p,k]+last_term[k]

        new_cluster_result[p] = np.argmin(distance)

    return new_cluster_result
```

#### h. Visualizing current clustering result and the whole steps

The *get\_current\_image* function first sets a color array recording different colors for visualization. A new array *colored\_cluster* is used to map each pixel to its corresponding clustering color, then reshape it to the same shape as the original image and use the *Image.fromarray* function to create an image from it.

```
def get_current_image(rows,cols,cluster_result):
    colors = np.array([[255,0,0],[0,255,0],[0,0,255],[0,0,0],[255,255,255],[0,255,255],[255,0,255],[255,255,0]])

    colored_cluster = np.zeros((rows*cols, 3))
    for idx in range(rows*cols):
        colored_cluster[idx,:] = colors[cluster_result[idx],:]
    img = colored_cluster.reshape(rows,cols,3)

    return Image.fromarray(np.uint8(img))
```

After the clustering, the whole clustering process is saved as a GIF to visualize the clustering process, and the clustering result is also saved.

```
title = filePath[:-4]+"_k_means_clusterNum_"+str(clusters)+"_mode_"+str(mode)
title += "_gs_"+str(gs)+"_gc_"+str(gc)

img_list[0].save(title+".gif",save_all=True,append_images=img_list[1:],optimize=False,loop=0,duration=300)
img_list[-1].save(title+"_result.png")
```

## B. Results

The results shown below are from the combination of two different possible images, image1.png and image2.png are shown below; three possible clustering counts from two to four; two possible  $\gamma_c$ , 0.0001 and 0.001; two possible  $\gamma_s$ , 0.0001 and 0.001, and two possible initialization methods for k-mean, the random method (mode = 0) and the k-means++ method (mode=1).

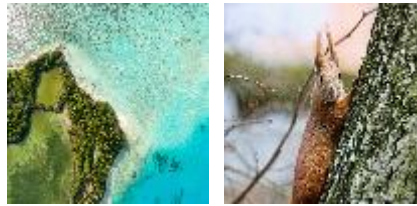


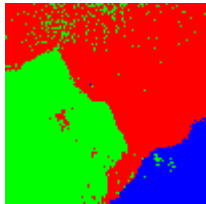
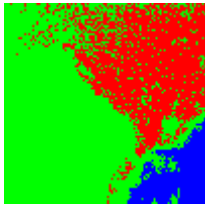
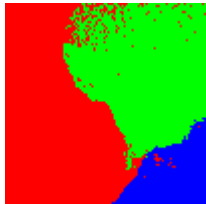
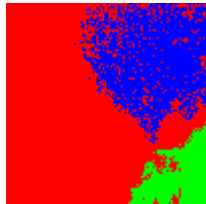
image1.png      image2.png

- a. Kernel k-means for image 1 with 2 clusters, different gamma parameters, and k-means initialization

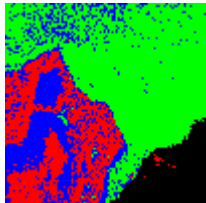
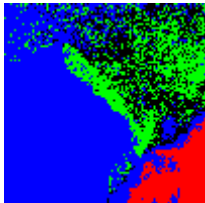
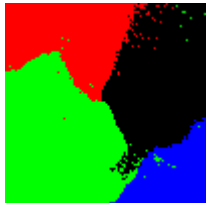
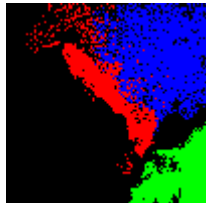
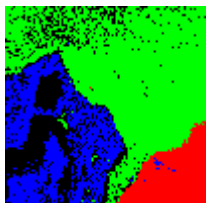
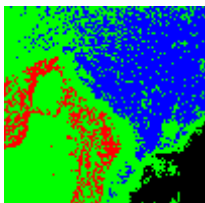

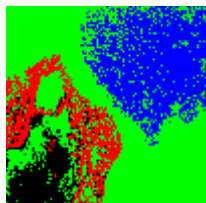
	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

- b. Kernel k-means for image 1 with 3 clusters, different gamma parameters, and k-means initialization








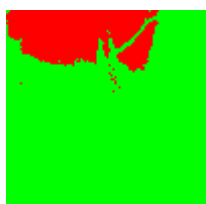
	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				

k-means plus plus				
-------------------------	---	---	--	---

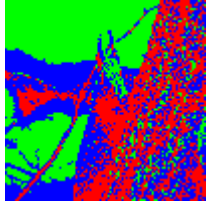
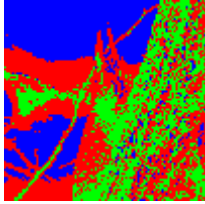
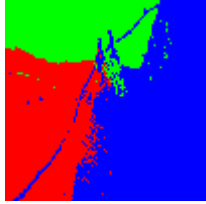
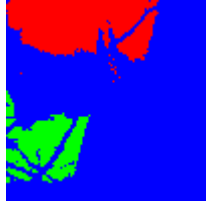
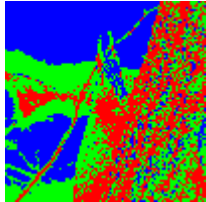
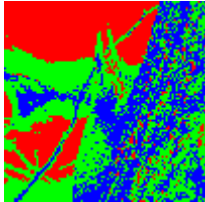
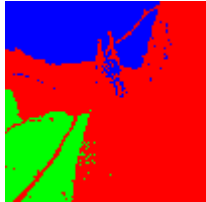
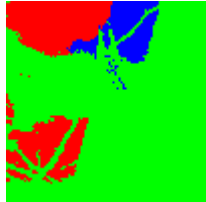
c. Kernel k-means for image 1 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

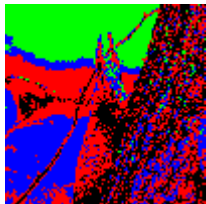
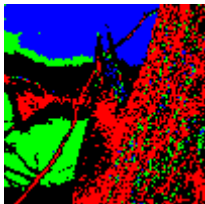
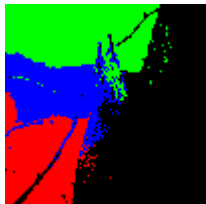
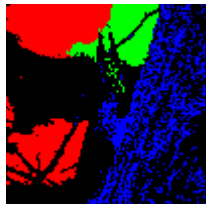
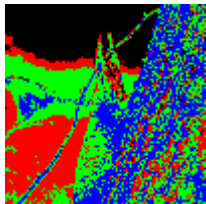
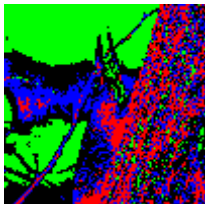
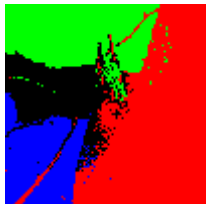
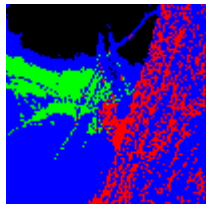
d. Kernel k-means for image 2 with 2 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

- e. Kernel k-means for image 2 with 3 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

- f. Kernel k-means for image 2 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

### C. Observations and Discussion (Comparison with the spectral clustering)

- a. Clustering results review

Judging by personal viewing perspective, kernel k-means clustering performs better cluster results than both ratio cut and normalized cut since kernel k-means at least the rough shape of different objects can be caught, the shape of the squirrel is more easily recognized in most of the kernel k-means results, and there is no case that almost the whole data points are categorized in the same cluster. However, since



it is an unsupervised method, there is still misclassification, and it is more apparent when it comes to a more complex case like image2.png.

b. Overall execution time review

The calculation time of the kernel in each clustering is not time-consuming compared to the time to calculate the eigenvalues and the eigenvectors. Still, the kernel k-means algorithm must compute the kernel and the equation mentioned before to get the distance for the next round of clustering, so it still takes time.

c. Comparison between different initialization methods of k-means clustering

In the kernel k-means algorithm, the random method clustering result does not perform poorly compared to the k-means++ result. They both separate the regions in most of the cases. The clustering with the k-means++ method is more straightforward to converge, so the execution time is less than that with the random method.

The initialization method heavily affects the initial clustering result, and k-means++ usually performs better than the random result. The k-means++ method may still pick the centers not far from each other, so there is a chance that the k-means++ method performs poorer than the random method.

d. Comparison between different hyper-parameters in kernel calculation and the number of clustering count

Different hyper-parameters in kernel calculation represent different clustering characteristics. The same hyper-parameters paired with varying numbers of clustering counts perform various effects.

For image 1, the left-down island is more recognizable under  $\gamma_c = 0.0001$ , while the ocean part is more recognizable under  $\gamma_c = 0.001$ . By increasing the number of clustering counts with the same  $\gamma_S$  and  $\gamma_c$  pair, the right-down darker ocean part is separate from the others, and sometimes the island's interior is also seen as a different part since the color may differ.

Image 2 is a more complicated case since the image's foreground and background show spectral and color similarity, and there are small objects with similar color similarity with each other, like thin branches. The foreground and the background are not nicely categorized into different clusters. The branches and the trunk are not easily separated from the forest-like scene in the background. The ears of the squeal are more easily distinguished from the background in all cases.

## II. Spectral Clustering

### A. Code explanation

#### a. Libraries, modules, and functions import

Libraries and modules imported into the program are shown below. The argparse library is imported for command-line options. The numpy module is imported for mathematical operations on arrays. The scipy.spatial.distance module is imported to compute the distance between two matrices. The PIL library is imported to read images and visualize the algorithm's results. The OS module is used for the OS routine. The time module is used to count the execution time. The *load\_img*, *calculate\_kernel*, *get\_centers*, and *get\_current\_image* functions are imported for further calculation.

```
import numpy as np
from scipy.spatial.distance import cdist
from PIL import Image
from kernel_k_means import load_img, calculate_kernel, get_centers, get_current_image
import os
import time
import matplotlib.pyplot as plt
```

#### b. The overall workflow of the program

The program is to perform both ratio cut and normalized cut algorithms below. The relaxation of the ratio cut is unnormalized spectral clustering.

##### Unnormalized spectral clustering

Input: Similarity matrix  $S \in \mathbb{R}^{n \times n}$ , number  $k$  of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let  $W$  be its weighted adjacency matrix.
- Compute the unnormalized Laplacian  $L$ .
- **Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of  $L$ .**
- Let  $U \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $u_1, \dots, u_k$  as columns.
- For  $i = 1, \dots, n$ , let  $y_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $U$ .
- Cluster the points  $(y_i)_{i=1, \dots, n}$  in  $\mathbb{R}^k$  with the  $k$ -means algorithm into clusters  $C_1, \dots, C_k$ .

Output: Clusters  $A_1, \dots, A_k$  with  $A_i = \{j \mid y_j \in C_i\}$ .

##### Normalized spectral clustering according to Ng, Jordan, and Weiss (2002)

Input: Similarity matrix  $S \in \mathbb{R}^{n \times n}$ , number  $k$  of clusters to construct.

- Construct a similarity graph by one of the ways described in Section 2. Let  $W$  be its weighted adjacency matrix.
- Compute the normalized Laplacian  $L_{\text{sym}}$ .
- **Compute the first  $k$  eigenvectors  $u_1, \dots, u_k$  of  $L_{\text{sym}}$ .**
- Let  $U \in \mathbb{R}^{n \times k}$  be the matrix containing the vectors  $u_1, \dots, u_k$  as columns.
- **Form the matrix  $T \in \mathbb{R}^{n \times k}$  from  $U$  by normalizing the rows to norm 1,**  
that is set  $t_{ij} = u_{ij} / (\sum_k u_{ik}^2)^{1/2}$ .
- For  $i = 1, \dots, n$ , let  $y_i \in \mathbb{R}^k$  be the vector corresponding to the  $i$ -th row of  $T$ .
- Cluster the points  $(y_i)_{i=1, \dots, n}$  with the  $k$ -means algorithm into clusters  $C_1, \dots, C_k$ .

Output: Clusters  $A_1, \dots, A_k$  with  $A_i = \{j \mid y_j \in C_i\}$ .

The file path of the input image, the total clustering number, the hyper-parameters for the kernel, different initialization methods for k-means, and different types of spectral clustering are set in each for loop. The image is loaded and used to calculate the kernel. The kernel is then used as a weighted adjacency matrix to calculate the Laplacian matrix and the corresponding eigenvector matrix. The initial centers are computed to perform the k-means algorithm to cluster the points into each cluster.

```
if __name__ == '__main__':
    gs=[1e-3,1e-4]
    gc=[1e-3,1e-4]
    img_list=["image1.png","image2.png"]
    for i in range(1,2):
        for clusterNum in range(2,5):
            for gs_idx in range(2):
                for gc_idx in range(2):
                    for m in range(2):
                        for c in range(2):
                            start = time.time()
                            print("image loading...")
                            img_c,img_s,rows,cols,channels = load_img(img_list[i])

                            print("kernel calculation...")
                            kernel = calculate_kernel(img_s,gs[gs_idx],img_c,gc[gc_idx])

                            print("clustering information calculation...")
                            U = get_clustering_info(img_list[i],clusterNum,gs[gs_idx],gc[gc_idx],m,c,kernel)

                            print("spectral clustering...")
                            centers_idx = get_centers(rows,cols,clusterNum,U,m)
                            centers = get_initial_centers(centers_idx,U)
                            perform_kmeans(rows,cols,clusterNum,img_list[i],m,centers,U,c,gs[gs_idx],gc[gc_idx])
                            end = time.time()
                            print("time: ",end-start,"s.")
                            print("=%*40,end = '\n\n')
```

c. Image loading

The input image is loaded and saved as an array by the *load\_img* function as it does in kernel k-means.

d. Kernel calculating

The defined kernel  $k(x, x')$  is computed by the *calculate\_kernel* function for the weighted adjacency matrix.

e. Calculating the needed Laplacian matrix and corresponding eigenvectors

The *get\_clustering\_info* function checks if the previous matrix file was saved. If the corresponding file does not exist, then the graph Laplacian and its eigenvector matrix for spectral clustering are calculated, then the eigenvector matrix *U* is saved.

```
def get_clustering_info(filePath,clusters,gs,gc,mode,cut,kernel):
    title = filePath[:-4]+"_clusterNum"+str(clusters)+"_cut_"+str(cut)
    title += "_gs_"+str(gs)+"_gc_"+str(gc)+".npy"
    if os.path.isfile(title)==True:
        print("file found.")
        clustering_info = np.load(title, allow_pickle=True)
        U = clustering_info.item().get('U')
    else:
        print("file not found.")
        L = compute_laplacian_matrix(kernel,cut)
        U = compute_eigenvector_matrix(L,clusters,cut)
        clustering_info = {'U' : U}
        np.save(title,clustering_info)
```

The Laplacian matrix  $L$  used in the ratio cut is unnormalized, while the normalized cut is the normalized Laplacian. The matrix  $U$  containing the non-zero eigenvalue corresponding eigenvectors is used in the ratio cut. In contrast, the matrix must be normalized to rows to norm 1 for the normalized cut.

By the *compute\_laplacian\_matrix* function, the Laplacian matrix  $L$  is first calculated as below.

$$L = D - W$$

Where  $D$  is the degree matrix for the image, and  $W$  represents the kernel matrix. As for the normalized cut, the normalized Laplacian matrix  $L_{sys}$  is calculated below.

$$L_{sys} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}}$$

```
def compute_laplacian_matrix(W,cut):
    D = np.zeros((W.shape))
    L = np.zeros((W.shape))
    D = np.diag(np.sum(W,axis=1))
    L = D-W

    if cut:
        for idx in range(D.shape[0]):
            D[idx,idx] = np.power(D[idx,idx],-0.5)
            L = D.dot(L).dot(D)

    return L
```

Then, in the *compute\_eigenvector\_matrix* function, the eigenvalues and the eigenvectors of  $L$  or  $L_{sys}$  are calculated by *numpy.linalg.eig()* function. The column in the eigenvectors is the eigenvectors corresponding to the eigenvalues. Eigenvectors with eigenvalues not bigger than zero are removed, then sorted by each corresponding eigenvalue, and the first  $k$  eigenvectors, where  $k$  represents the count of the clusters, thus forming the matrix  $U$ . For the normalized cut, the rows of  $U$  are then normalized to norm 1.

```
def compute_eigenvector_matrix(L, clusters, cut):
    eigenvalues, eigenvectors = np.linalg.eig(L)
    eigenvectors = eigenvectors.T

    sort_idx = eigenvalues.argsort()
    sort_idx = sort_idx[eigenvalues[sort_idx]>0]

    U = eigenvectors[sort_idx[:clusters]].T
    if cut:
        U_row_sum = np.sum(U, axis=1)
        for r in range(U.shape[0]):
            U[r,:] /= U_row_sum[r]

    return U
```

f. Getting initial centers for the k-means algorithm

The *get\_centers* function is used to get the indices of the image data coordinates by either the random method or the k-means++ method, as the user asked; it is the same as in kernel k-means. The indices are then used to calculate the corresponding index in the eigenspace to get initial center points in the *get\_initial\_centers* function.

```
def get_initial_centers(centers, U):
    center_list = []
    for k in range(len(centers)):
        center_list.append(U[centers[k],:])
    return np.array(center_list)
```

g. K-means algorithm

The *perform\_kmeans* function performs the k-means algorithm after getting the initial center. The maximum iteration number and the threshold are set. While the maximum iteration number is not reached yet, the iteration count parameter will be renewed. The new clustering result is calculated using the *U* matrix and centers by the *get\_new\_cluster* function, and the *get\_new\_centers* function renews the new center points. If the L2 norm of the new and old clustering results is smaller than the threshold, the computation is seen as coverage and stop; otherwise, set the latest result as the old one and visualize and save it. After the clustering, the whole clustering process is visualized and saved.

```
def perform_kmeans(rows,cols,clusters,filePath,mode,centers,U,cut,gs,gc):
    max_iter = 50
    threshold = 1e-2
    img_list = []
    cluster_result = np.zeros(rows*cols, dtype=int)-1

    iter = 0
    while iter < max_iter:
        print(iter)
        iter += 1
        new_cluster_result = get_new_cluster(rows,cols,clusters,U,centers)
        if iter!=1 and (np.linalg.norm((new_cluster_result-cluster_result))<threshold):
            break
        cluster_result = new_cluster_result.copy()
        centers = get_new_centers(clusters,U,cluster_result)
        img_list.append(get_current_image(rows,cols,cluster_result))

    title = filePath[:-4]+ "_spectral_clusterNum_"+str(clusters)+"_cut_"+str(cut)+"_mode_"+str(mode)
    title += "_gs_"+str(gs)+"_gc_"+str(gc)
    img_list[0].save(title+".gif",save_all=True,append_images=img_list[1:],optimize=False,loop=0,duration=300)
    img_list[-1].save(title+"_result.png")
    if clusters == 2 or clusters == 3:
        visualize_eigenspace(U,cluster_result,clusters,title)

    return
```

The `get_new_cluster` function calculates the distance between the current pixel in the eigenspace and each center point for each pixel in the image. Then, the pixel will be categorized into the same cluster with a minimal distance.

```
def get_new_cluster(rows,cols,clusters,U,centers):
    pixels = rows*cols
    cluster_result = np.zeros(pixels, dtype=int)

    for idx in range(pixels):
        distance = np.zeros(clusters)
        for k in range(clusters):
            distance[k] = np.linalg.norm(U[idx]-centers[k])
        cluster_result[idx] = np.argmin(distance)

    return cluster_result
```

The `get_new_centers` function calculates the average coordinate position in the eigenspace of each cluster and is renewed as the new center point.

```
def get_new_centers(clusters,U,cluster_result):
    center_list = []
    for k in range(clusters):
        category_k = U[cluster_result==k]
        center_list.append(np.average(category_k,axis=0))

    return np.array(center_list)
```

#### h. Visualizing the current clustering result and all the steps.

The `get_current_image` function visualizes the current cluster result array to an image as it does in kernel k-means.

After the clustering, the whole clustering process is saved as a GIF to visualize the clustering process, and the clustering result is also saved.

```

title = filePath[:-4]+"_spectral_clusterNum_"+str(clusters)+"_cut_"+str(cut)+"_mode_"+str(mode)
title += " _gs_"+str(gs)+"_gc_"+str(gc)
img_list[0].save(title+".gif", save_all=True, append_images=img_list[1:], optimize=False, loop=0, duration=300)
img_list[-1].save(title+"_result.png")
if clusters == 2 or clusters == 3:
    visualize_eigenspace(U, cluster_result, clusters, title)

```

i. Visualizing data points in the eigenspace of graph Laplacian

If the clustering count is two or three, visualize the clustering result in the eigenspace using the *visualize\_eigenspace* function. The color for each cluster is set. For the result with two clusters, plot a 2-dimensional scatter plot, and for the result with three clusters, plot a 3-dimensional scatter plot to visualize the distribution of the data points in the eigenspace.

```

def visualize_eigenspace(U, result, clusters, title):
    colors = ['r', 'g', 'b']
    plt.clf()
    if clusters == 2:
        plt.xlabel("Feature #1")
        plt.ylabel("Feature #2")
        for p in range(U.shape[0]):
            plt.scatter(U[p,0], U[p,1], c=colors[result[p]], s=2)
        plt.savefig(title+"_eigenspace.png")
    else:
        fig=plt.figure()
        ax = plt.axes(projection="3d")
        ax.set_xlabel("Feature #1")
        ax.set_ylabel("Feature #2")
        ax.set_zlabel("Feature #3")
        for p in range(U.shape[0]):
            ax.scatter(U[p,0], U[p,1], np.real(U[p,2]), c=colors[result[p]], s=2)
        plt.savefig(title+"_eigenspace.png")
    return

```

## B. Results

The clustering results shown below are from the combination of two different possible images, image1.png and image2.png are shown below; three possible clustering counts from two to four; two possible  $\gamma_c$ , 0.0001 and 0.001; two possible  $\gamma_c$ , 0.0001 and 0.001; two possible initialization methods for k-mean, the random method(mode = 0) and the k-means++ method(mode=1), and two spectral clustering methods, the ratio cut and the normalized cut.

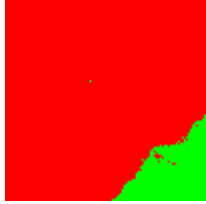
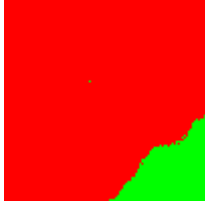
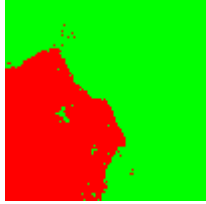
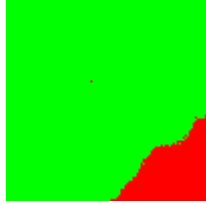
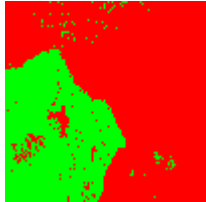
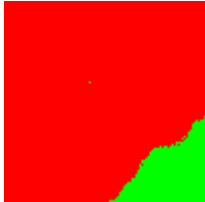
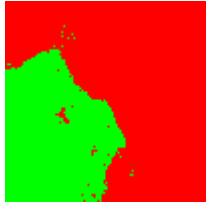
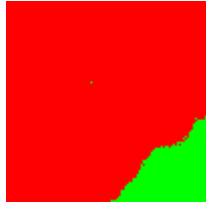


image1.png





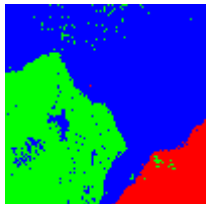





image2.png

- a. Ratio cut for image 1 with 2 clusters, different gamma parameters, and k-means initialization

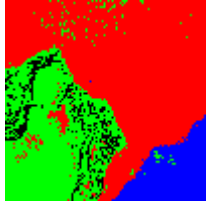
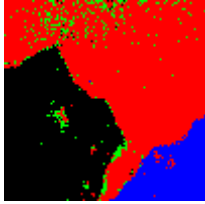
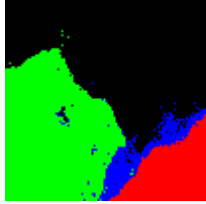
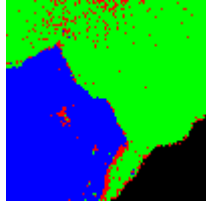
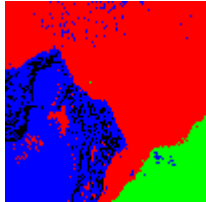

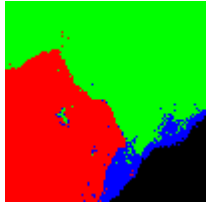
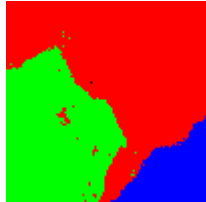
	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

- b. Ratio cut for image 1 with 3 clusters, different gamma parameters, and k-means initialization


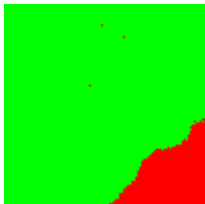
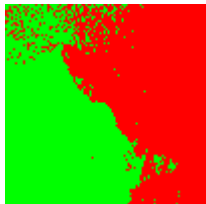
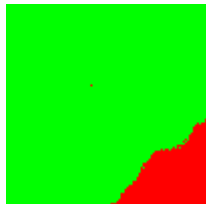


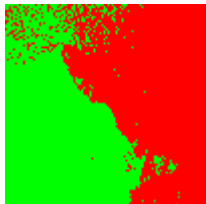
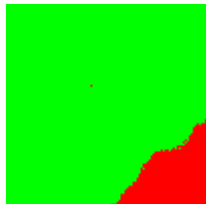
	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				



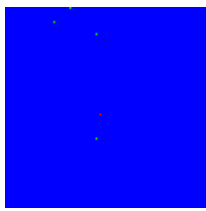
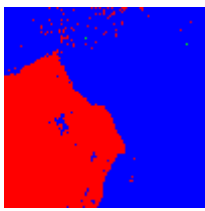
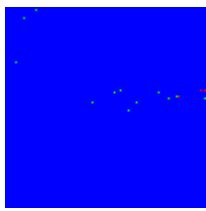
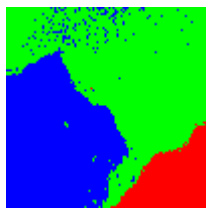
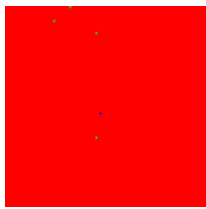
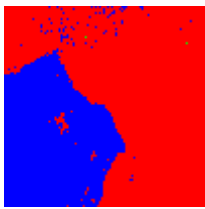

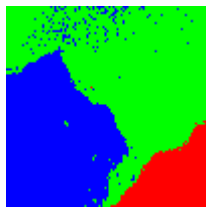
- c. Ratio cut for image 1 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

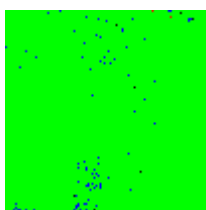
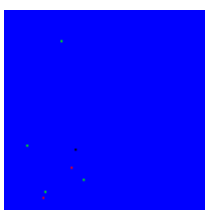
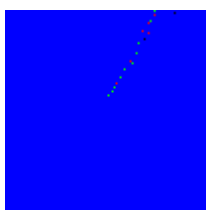
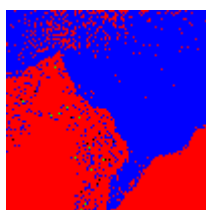
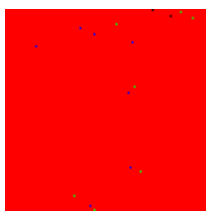
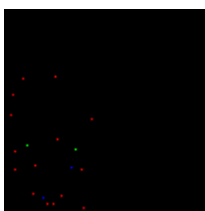
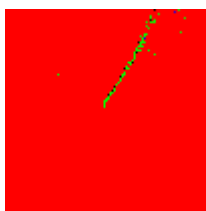
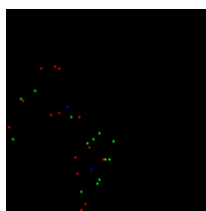
- d. Normalized cut for image 1 with 2 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

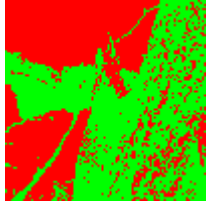
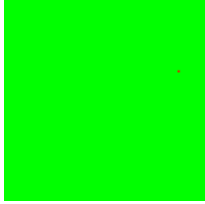
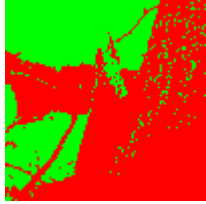
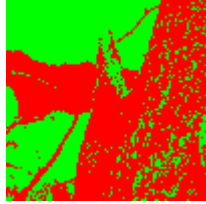
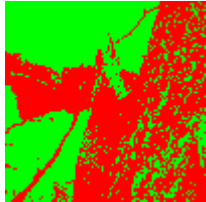

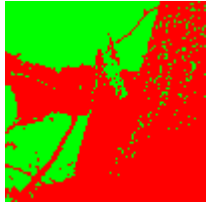
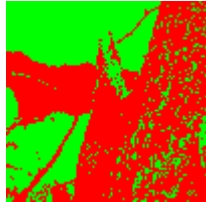
- e. Normalized cut for image 1 with 3 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

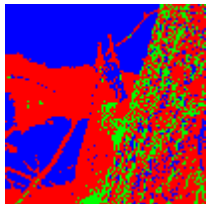
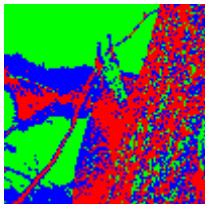
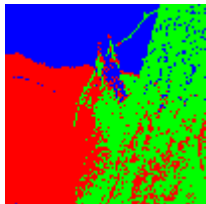
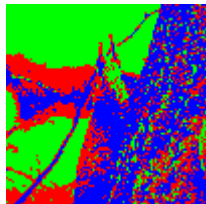
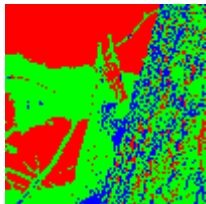
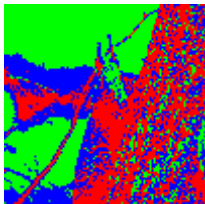
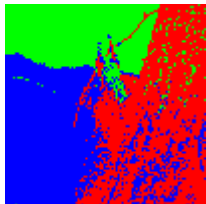
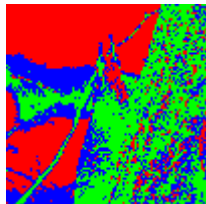
- f. Normalized cut for image 1 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

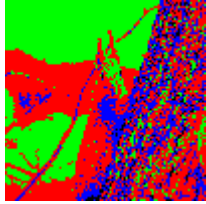
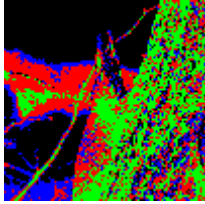
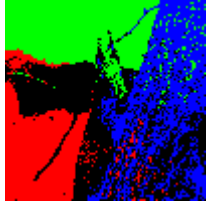
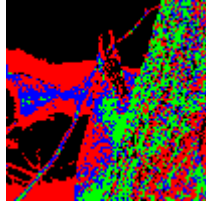
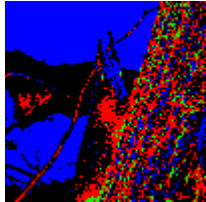
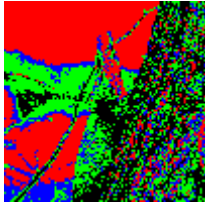
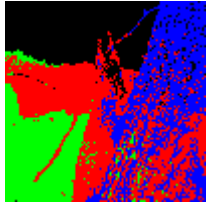
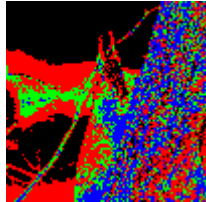
- g. Ratio cut for image 2 with 2 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

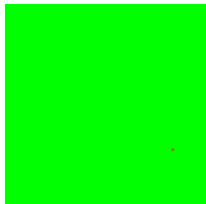
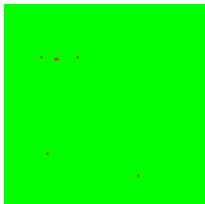
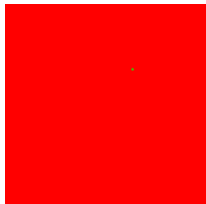
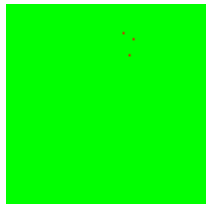
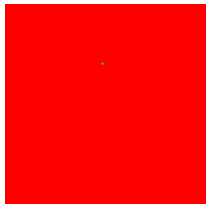


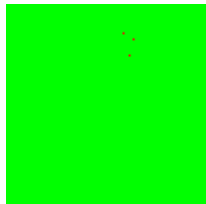
- h. Ratio cut for image 2 with 3 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				


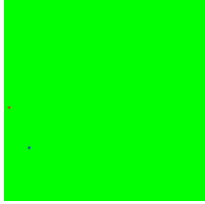

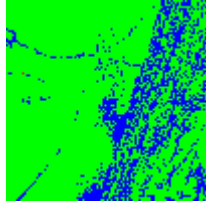

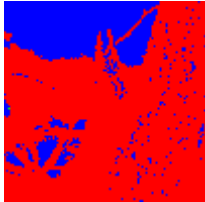

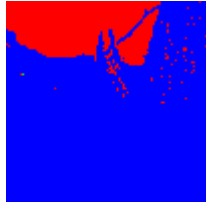
- i. Ratio cut for image 2 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

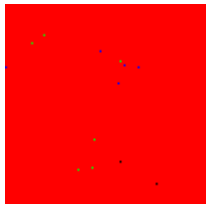
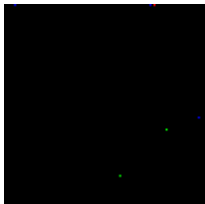
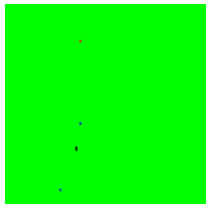
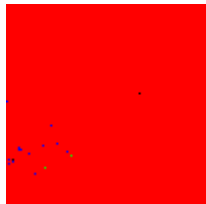
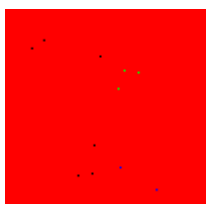
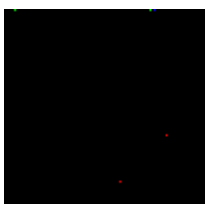
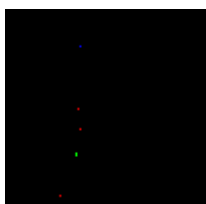
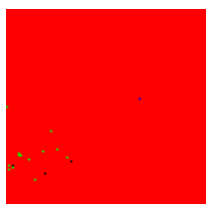
- j. Normalized cut for image 2 with 2 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

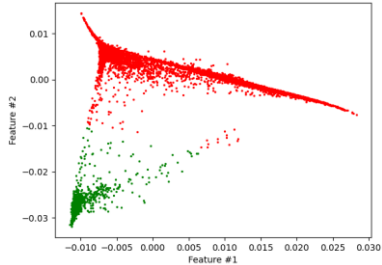
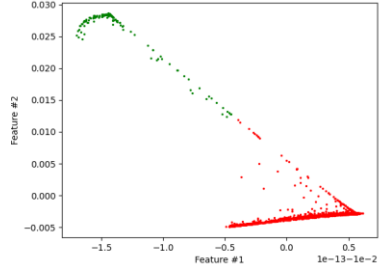
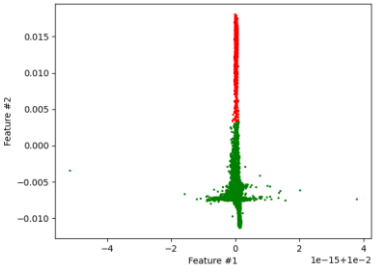
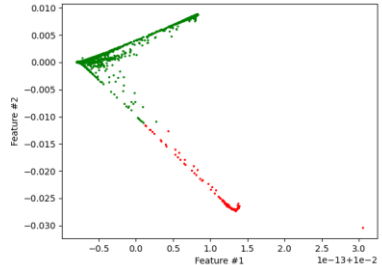
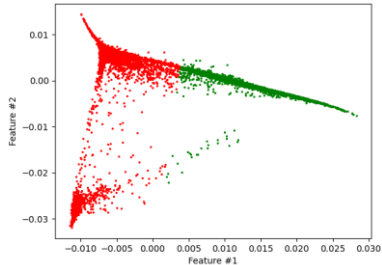
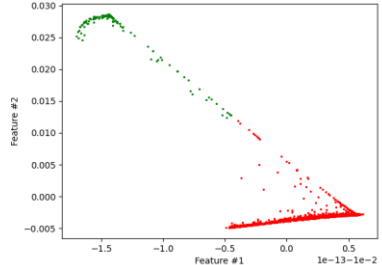
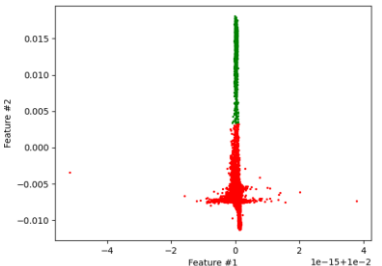
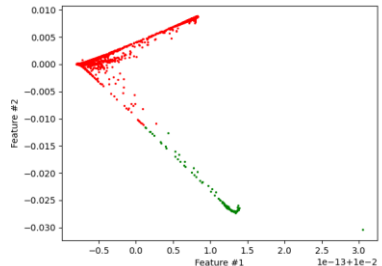
- k. Normalized cut for image 2 with 3 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

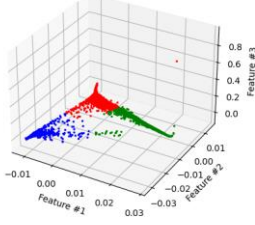
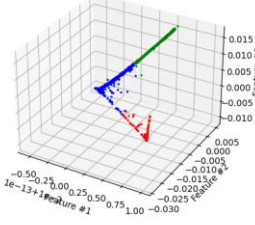
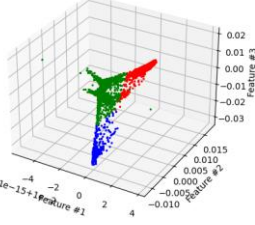
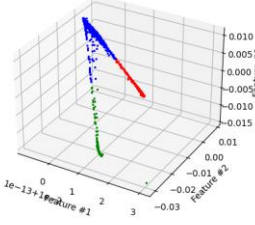
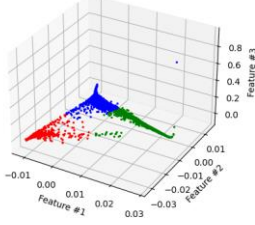
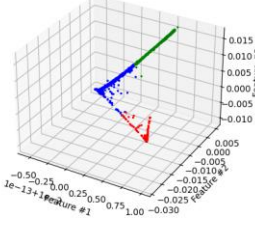
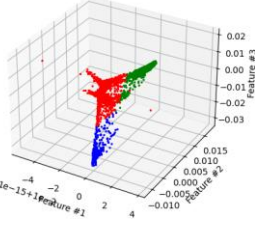
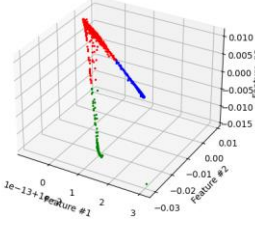
- l. Normalized cut for image 2 with 4 clusters, different gamma parameters, and k-means initialization

	$\gamma_s = 0.0001$ $\gamma_c = 0.0001$	$\gamma_s = 0.0001$ $\gamma_c = 0.001$	$\gamma_s = 0.001$ $\gamma_c = 0.0001$	$\gamma_s = 0.001$ $\gamma_c = 0.001$
random				
k-means plus plus				

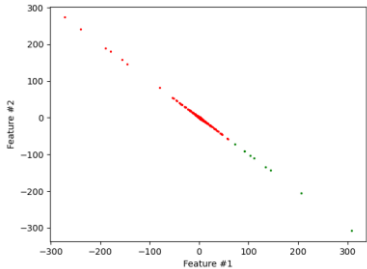
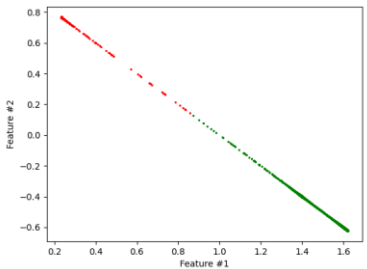
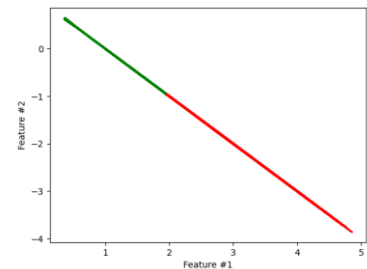
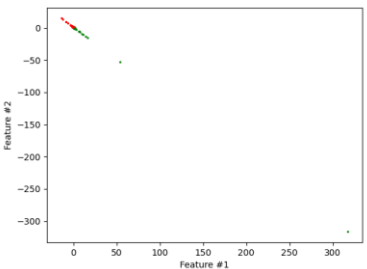
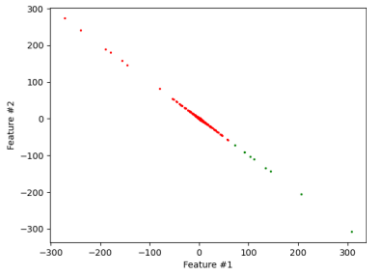
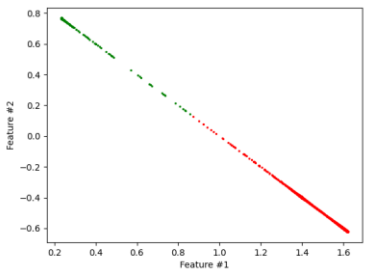
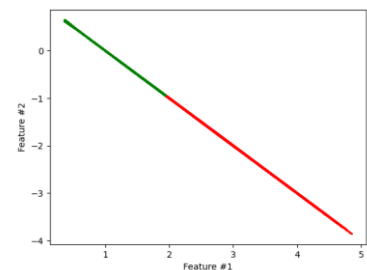
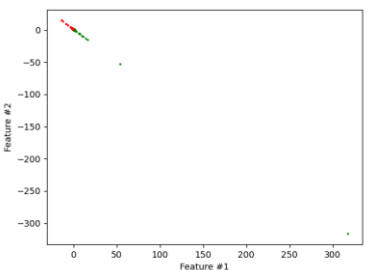
- m. Coordinates of ratio cut for image 1 with 2 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

- n. Coordinates of ratio cut for image 1 with 3 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

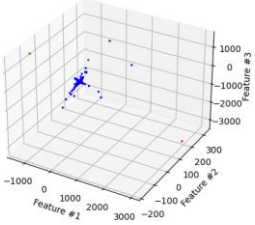
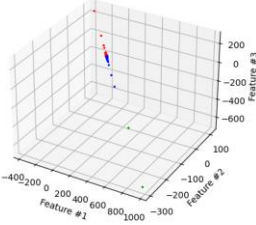
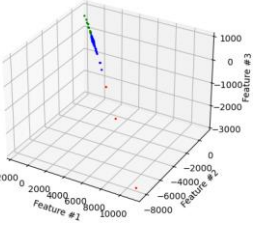
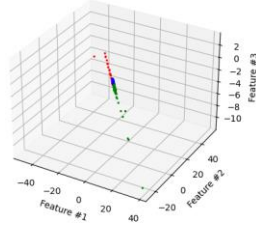
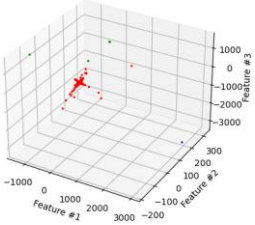
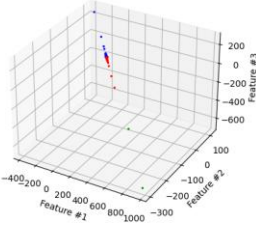
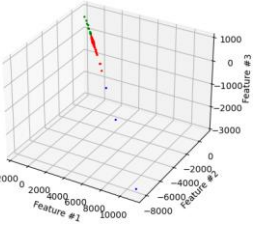
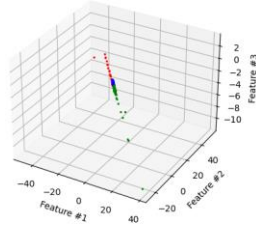
		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

- o. Coordinates of normalized cut for image 1 with 2 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

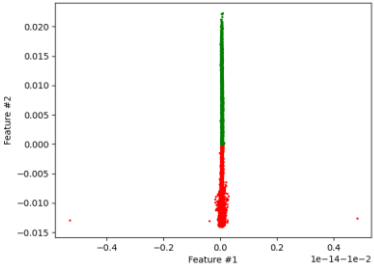
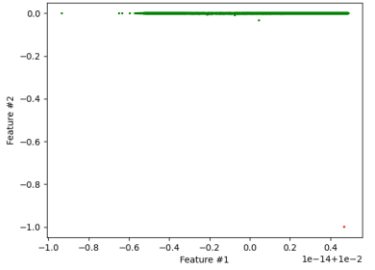
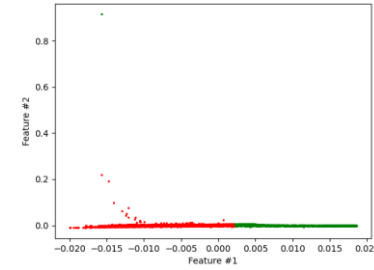
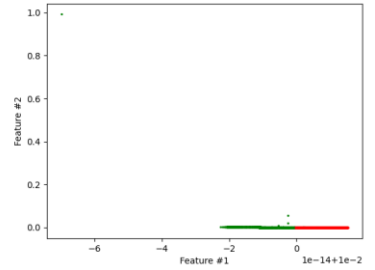
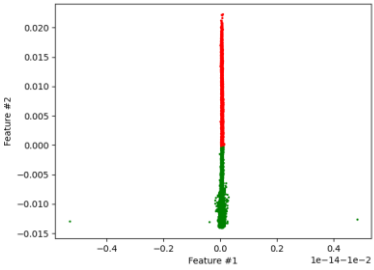
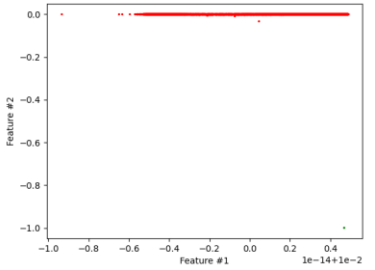
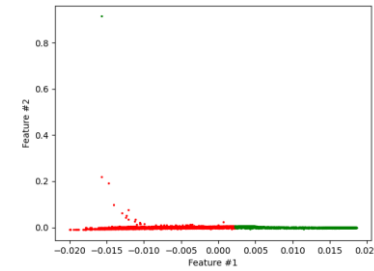
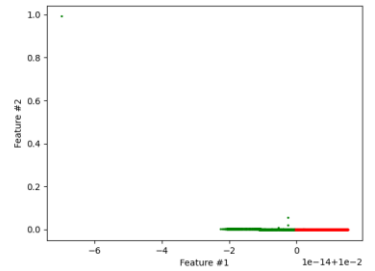
		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		



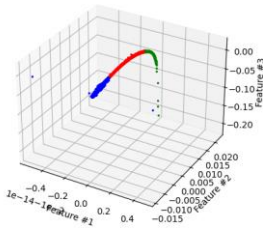
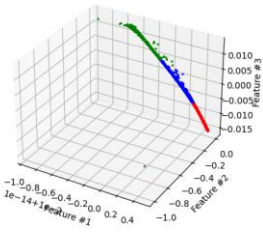
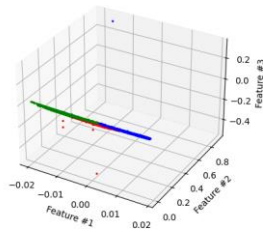
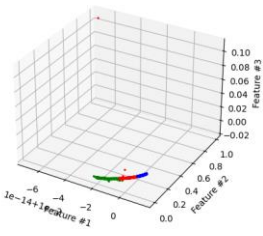
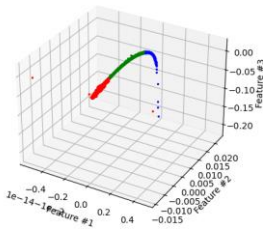
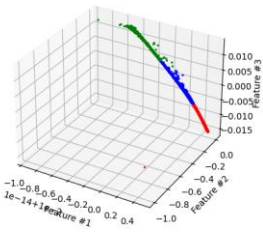
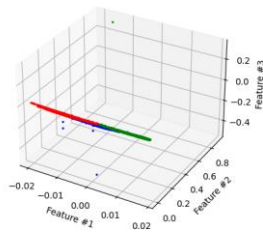
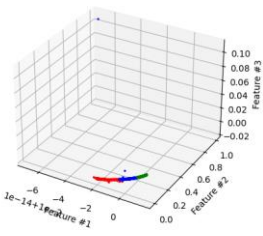
- p. Coordinates of normalized cut for image 1 with 3 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

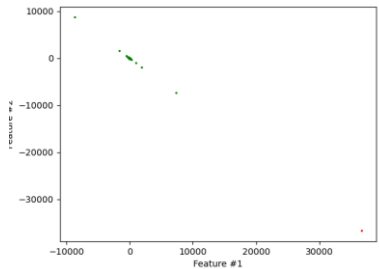
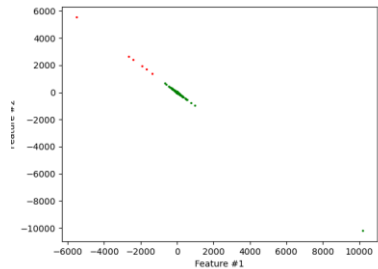
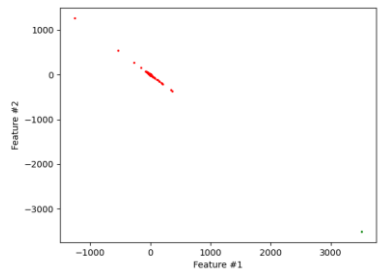
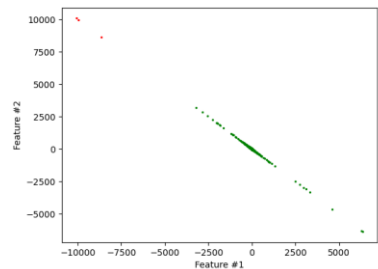
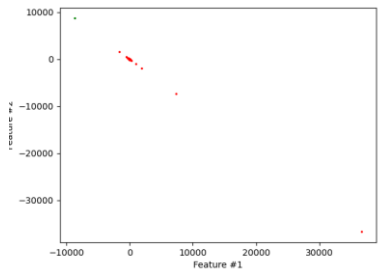
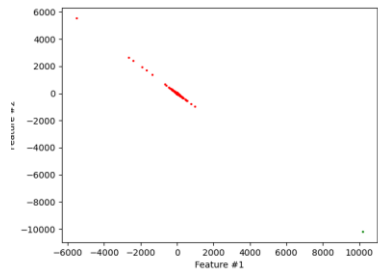
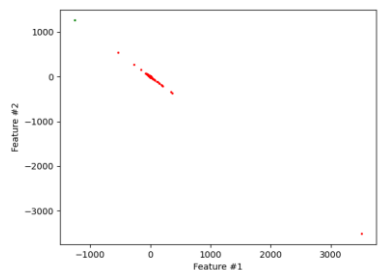
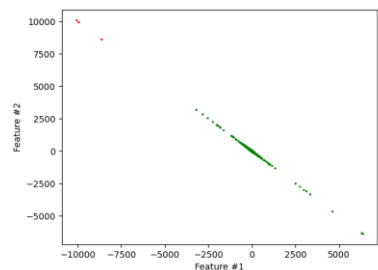
q. Coordinates of ratio cut for image 2 with 2 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

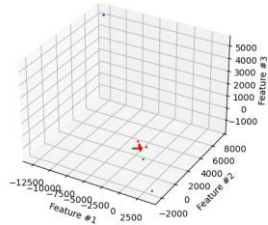
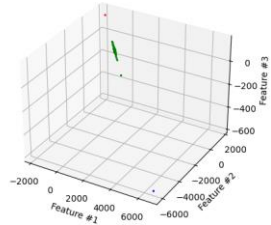
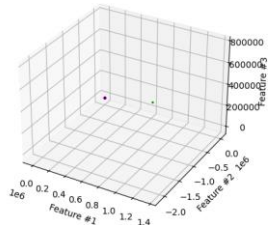
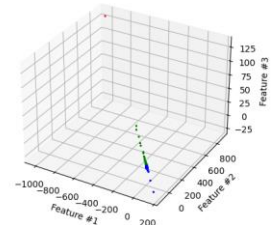
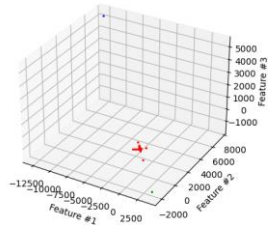
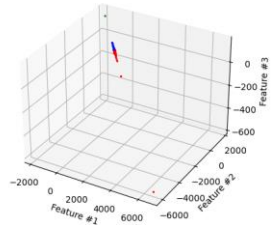
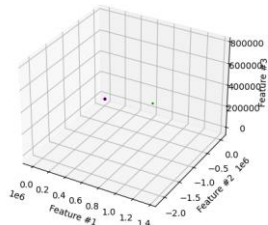
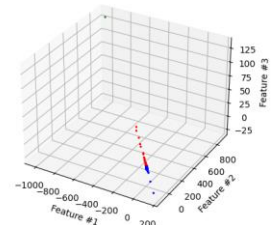
- r. Coordinates of ratio cut for image 2 with 3 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

- s. Coordinates of normalized cut for image 2 with 2 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4 (0.0001)$	$\gamma_c = 1e-3 (0.001)$
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

- t. Coordinates of normalized cut for image 2 with 3 clusters, different gamma parameters, and k-means initialization in the eigenspace of graph Laplacian

		$\gamma_c = 1e-4$ (0.0001)	$\gamma_c = 1e-3$ (0.001)
random	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		
k-means plus plus	$\gamma_s = 1e-4$		
	$\gamma_s = 1e-3$		

### C. Observations and Discussion (compared to the kernel k-means)

- a. Clustering results review

The clustering result is more unstable, and there is potential to categorize almost the whole image into the same cluster despite the separated initial centers. The better clustering result shows similarity with that in the kernel k-means. The ratio cut captures a more precise boundary than other methods do.

b. Overall execution time review

Although the first-time calculation of graph Laplacian is highly time-consuming, it usually takes about 200 to 400 seconds in ratio cut and about 500 to 700 seconds in normalized cut, where the normalization also takes time. After saving the matrix of graph Laplacian, the time for a round can be reduced to about 30 seconds. Furthermore, without constructing the graph Laplacian and visualization in the eigenspace of the graph Laplacian, the time of a spectral clustering round can even reduce the time to 5 seconds.

c. The visualization of the eigenspace of graph Laplacian

The result shows that data points in the same clusters are close in the eigenspace of graph Laplacian in most cases, while in some cases, outliers in the eigenspace can be seen, thus causing a massive range of the coordinates. The shape of data distribution of the same image in the eigenspace is quite different between the ratio cut and the normalized cut. Besides different initialization methods, the visualization of the same clustering number and kernel hyper-parameter shows the same shape since the graph Laplacian is the same.

d. comparison between ratio cut and normalized cut

The ratio cut shows a more stable clustering ability than the normalized cut since there are cases in which almost every data point is assigned to the same cluster regardless of the initial method.

e. Comparison between different initialization methods of k-means clustering

There are not many noticeable differences in the ratio cut result. However, there are cases where the k-means++ method results in better clustering in the normalized cut. More experiments are needed to get more information.

f. Comparison between different hyper-parameters in kernel calculation and the number of clustering count

Observations are similar to the kernel k-means part. The number clustering of three is considered a better choice for the two given images since the segment result is more acceptable. In contrast, four clusterings may over-spilt, and two clusters usually don't represent the actual regions.